



下载APP



11 基础篇 | TCP连接的建立和断开受哪些系统配置影响？

2020-09-12 邵亚方

Linux内核技术实战课

[进入课程 >](#)**讲述：邵亚方**

时长 16:19 大小 14.95M



你好，我是邵亚方。

如果你做过 Linux 上面网络相关的开发，或者分析过 Linux 网络相关的问题，那你肯定吐槽过 Linux 系统里面让人眼花缭乱的各种配置项，应该也被下面这些问题困扰过：

Client 为什么无法和 Server 建立连接呢？

三次握手都完成了，为什么会收到 Server 的 reset 呢？

建立 TCP 连接怎么会消耗这么多时间？

系统中为什么会有这么多处于 time-wait 的连接？该怎么处理？

系统中为什么会有这么多 close-wait 的连接？



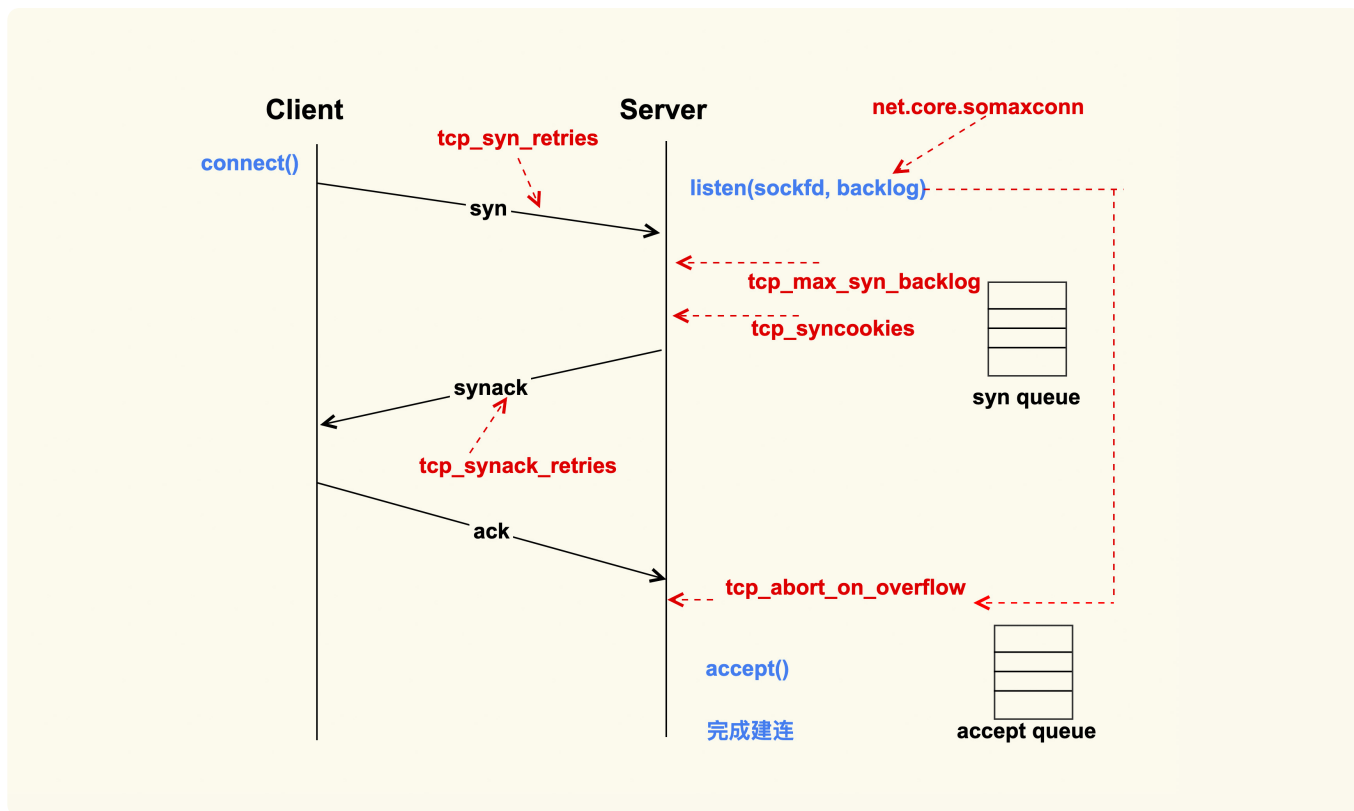
针对我的业务场景，这么多的网络配置项，应该要怎么配置呢？

.....

因为网络这一块涉及到的场景太多了，Linux 内核需要去处理各种各样的网络场景，不同网络场景的处理策略也会有所不同。而 Linux 内核的默认网络配置可能未必会适用我们的场景，这就可能导致我们的业务出现一些莫名其妙的行为。

所以，要想让业务行为符合预期，你需要了解 Linux 的相关网络配置，让这些配置更加适用于你的业务。Linux 中的网络配置项是非常多的，为了让你更好地了解它们，我就以最常用的 TCP/IP 协议为例，从一个网络连接是如何建立起来的以及如何断开的来开始讲起。

TCP 连接的建立过程会受哪些配置项的影响？



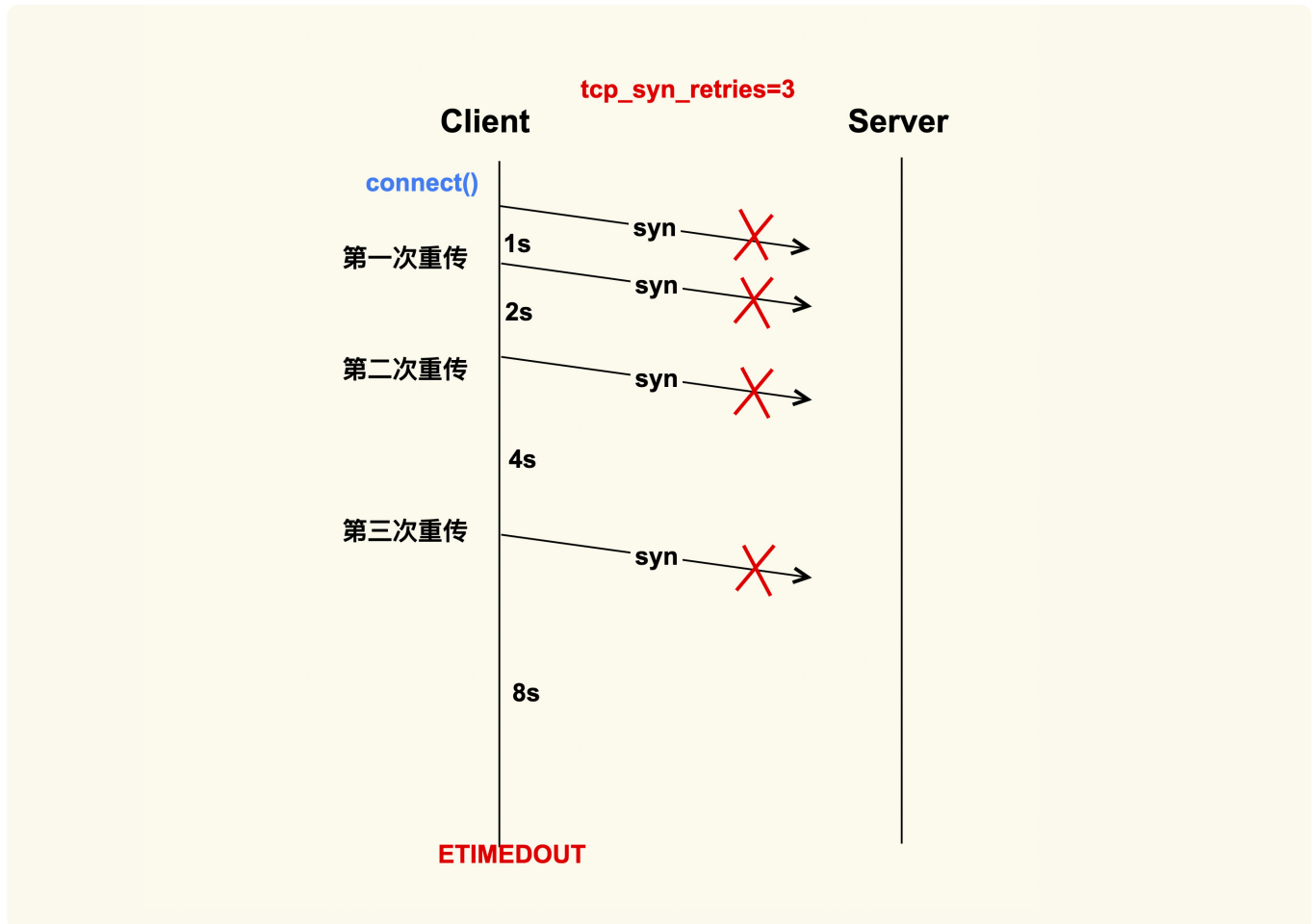
TCP建连过程

上图就是一个 TCP 连接的建立过程。TCP 连接的建立是一个从 Client 侧调用 `connect()`，到 Server 侧 `accept()` 成功返回的过程。你可以看到，在整个 TCP 建立连接的过程中，各个行为都有配置选项来进行控制。

Client 调用 `connect()` 后，Linux 内核就开始进行三次握手。

首先 Client 会给 Server 发送一个 SYN 包，但是该 SYN 包可能会在传输过程中丢失，或者因为其他原因导致 Server 无法处理，此时 Client 这一侧就会触发超时重传机制。但是也不能一直重传下去，重传的次数也是有限制的，这就是 `tcp_syn_retries` 这个配置项来决定的。

假设 `tcp_syn_retries` 为 3，那么 SYN 包重传的策略大致如下：



tcp_syn_retries示意图

在 Client 发出 SYN 后，如果过了 1 秒，还没有收到 Server 的响应，那么就会进行第一次重传；如果经过 2s 的时间还没有收到 Server 的响应，就会进行第二次重传；一直重传 `tcp_syn_retries` 次。

对于 `tcp_syn_retries` 为 3 而言，总共会重传 3 次，也就是说从第一次发出 SYN 包后，会一直等待 $(1 + 2 + 4 + 8)$ 秒，如果还没有收到 Server 的响应，`connect()` 就会产生 `ETIMEDOUT` 的错误。

`tcp_syn_retries` 的默认值是 6，也就是说如果 SYN 一直发送失败，会在 $(1 + 2 + 4 + 8 + 16 + 32 + 64)$ 秒，即 127 秒后产生 `ETIMEDOUT` 的错误。

我们在生产环境上就遇到过这种情况，Server 因为某些原因被下线，但是 Client 没有被通知到，所以 Client 的 connect() 被阻塞 127s 才去尝试连接一个新的 Server，这么长的超时等待时间对于应用程序而言是很难接受的。

所以通常情况下，我们都会将数据中心内部服务器的 tcp_syn_retries 给调小，这里推荐设置为 2，来减少阻塞的时间。因为对于数据中心而言，它的网络质量是很好的，如果得不到 Server 的响应，很可能是 Server 本身出了问题。在这种情况下，Client 及早地去尝试连接其他的 Server 会是一个比较好的选择，所以对于客户端而言，一般都会做如下调整：

```
net.ipv4.tcp_syn_retries = 2
```

有些情况下 1s 的阻塞时间可能都很久，所以有的时候也会将三次握手的初始超时时间从默认值 1s 调整为一个较小的值，比如 100ms，这样整体的阻塞时间就会小很多。这也是数据中心内部经常进行一些网络优化的原因。

如果 Server 没有响应 Client 的 SYN，除了我们刚才提到的 Server 已经不存在了这种情况外，还有可能是因为 Server 太忙没有来得及响应，或者是 Server 已经积压了太多的半连接 (incomplete) 而无法及时去处理。

半连接，即收到了 SYN 后还没有回复 SYNACK 的连接，Server 每收到一个新的 SYN 包，都会创建一个半连接，然后把该半连接加入到半连接队列 (syn queue) 中。syn queue 的长度就是 tcp_max_syn_backlog 这个配置项来决定的，当系统中积压的半连接个数超过了该值后，新的 SYN 包就会被丢弃。对于服务器而言，可能瞬间会有非常多的新建连接，所以我们可以适当地调大该值，以免 SYN 包被丢弃而导致 Client 收不到 SYNACK：

```
net.ipv4.tcp_max_syn_backlog = 16384
```

Server 中积压的半连接较多，也有可能是因为有些恶意的 Client 在进行 SYN Flood 攻击。典型的 SYN Flood 攻击如下：Client 高频地向 Server 发 SYN 包，并且这个 SYN 包的源 IP 地址不停地变换，那么 Server 每次接收到一个新的 SYN 后，都会给它分配一个半连接，Server 的 SYNACK 根据之前的 SYN 包找到的是错误的 Client IP，所以也就无法

收到 Client 的 ACK 包，导致无法正确建立 TCP 连接，这就会让 Server 的半连接队列耗尽，无法响应正常的 SYN 包。

为了防止 SYN Flood 攻击，Linux 内核引入了 SYN Cookies 机制。SYN Cookie 的原理是什么样的呢？

在 Server 收到 SYN 包时，不去分配资源来保存 Client 的信息，而是根据这个 SYN 包计算出一个 Cookie 值，然后将 Cookie 记录到 SYNACK 包中发送出去。对于正常的连接，该 Cookies 值会随着 Client 的 ACK 报文被带回来。然后 Server 再根据这个 Cookie 检查这个 ACK 包的合法性，如果合法，才去创建新的 TCP 连接。通过这种处理，SYN Cookies 可以防止部分 SYN Flood 攻击。所以对于 Linux 服务器而言，推荐开启 SYN Cookies：

```
net.ipv4.tcp_syncookies = 1
```

Server 向 Client 发送的 SYNACK 包也可能会被丢弃，或者因为某些原因而收不到 Client 的响应，这个时候 Server 也会重传 SYNACK 包。同样地，重传的次数也是由配置选项来控制的，该配置选项是 `tcp_synack_retries`。

`tcp_synack_retries` 的重传策略跟我们在前面讲的 `tcp_syn_retries` 是一致的，所以我们就不再画图来讲解它了。它在系统中默认是 5，对于数据中心的服务器而言，通常都不需要这么大的值，推荐设置为 2：

```
net.ipv4.tcp_synack_retries = 2
```

Client 在收到 Server 的 SYNACK 包后，就会发出 ACK，Server 收到该 ACK 后，三次握手就完成了，即产生了一个 TCP 全连接（complete），它会被添加到全连接队列（accept queue）中。然后 Server 就会调用 `accept()` 来完成 TCP 连接的建立。

但是，就像半连接队列（syn queue）的长度有限制一样，全连接队列（accept queue）的长度也有限制，目的就是为了防止 Server 不能及时调用 `accept()` 而浪费太多的系统资源。

全连接队列（accept queue）的长度是由 `listen(sockfd, backlog)` 这个函数里的 `backlog` 控制的，而该 `backlog` 的最大值则是 `somaxconn`。`somaxconn` 在 5.4 之前的内核中，默认都是 128（5.4 开始调整为了默认 4096），建议将该值适当调大一些：

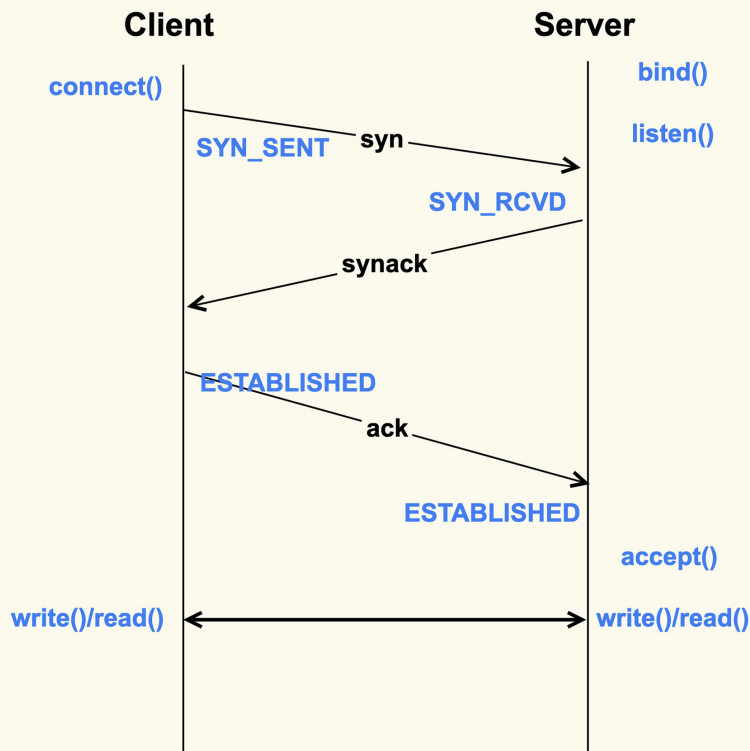
```
net.core.somaxconn = 16384
```

当服务器中积压的全连接个数超过该值后，新的全连接就会被丢弃掉。Server 在将新连接丢弃时，有的时候需要发送 `reset` 来通知 Client，这样 Client 就不会再次重试了。不过，默认行为是直接丢弃不去通知 Client。至于是否需要给 Client 发送 `reset`，是由 `tcp_abort_on_overflow` 这个配置项来控制的，该值默认为 0，即不发送 `reset` 给 Client。推荐也是将该值配置为 0：

```
net.ipv4.tcp_abort_on_overflow = 0
```

这是因为，Server 如果来不及 `accept()` 而导致全连接队列满，这往往是由瞬间有大量新建连接请求导致的，正常情况下 Server 很快就能恢复，然后 Client 再次重试后就可以建连成功了。也就是说，将 `tcp_abort_on_overflow` 配置为 0，给了 Client 一个重试的机会。当然，你可以根据你的实际情况来决定是否要使能该选项。

`accept()` 成功返回后，一个新的 TCP 连接就建立完成了，TCP 连接进入到了 `ESTABLISHED` 状态：

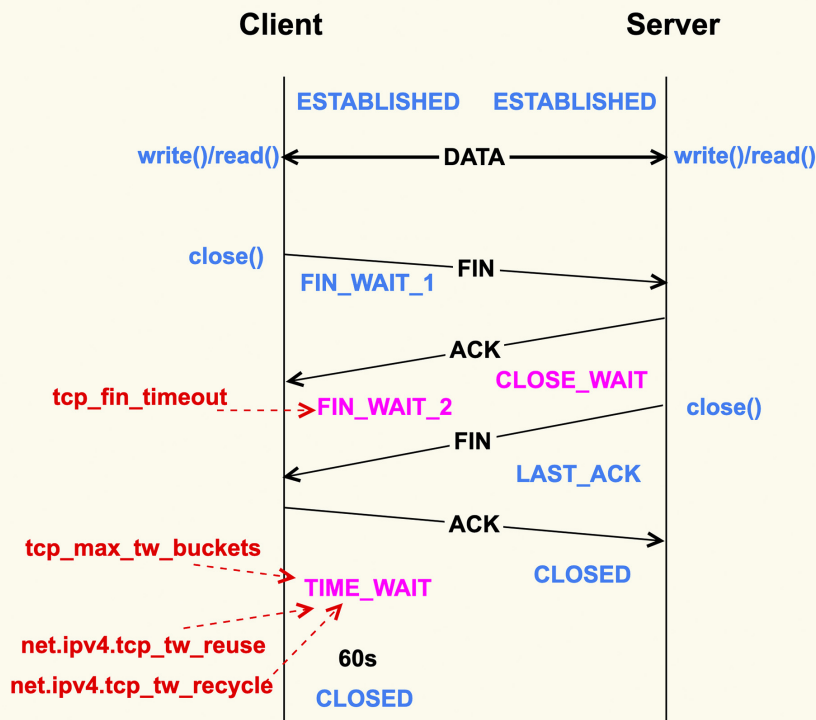


TCP状态转换

上图就是从 Client 调用 `connect()`，到 Server 侧 `accept()` 成功返回这一过程中的 TCP 状态转换。这些状态都可以通过 `netstat` 或者 `ss` 命令来看。至此，Client 和 Server 两边就可以正常通信了。

接下来，我们看下 TCP 连接断开过程中会受哪些系统配置项的影响。

TCP 连接的断开过程会受哪些配置项的影响？



TCP的四次挥手

如上所示，当应用程序调用 `close()` 时，会向对端发送 `FIN` 包，然后会接收 `ACK`；对端也会调用 `close()` 来发送 `FIN`，然后本端也会向对端回 `ACK`，这就是 TCP 的四次挥手过程。

首先调用 `close()` 的一侧是 active close（主动关闭）；而接收到对端的 `FIN` 包后再调用 `close()` 来关闭的一侧，称之为 passive close（被动关闭）。在四次挥手的过程中，有三个 TCP 状态需要额外关注，就是上图中深红色的那三个状态：主动关闭方的 `FIN_WAIT_2` 和 `TIME_WAIT`，以及被动关闭方的 `CLOSE_WAIT` 状态。除了 `CLOSE_WAIT` 状态外，其余两个状态都有对应的系统配置项来控制。

我们首先来看 `FIN_WAIT_2` 状态，TCP 进入到这个状态后，如果本端迟迟收不到对端的 `FIN` 包，那就会一直处于这个状态，于是就会一直消耗系统资源。Linux 为了防止这种资源的开销，设置了这个状态的超时时间 `tcp_fin_timeout`，默认为 60s，超过这个时间后就会自动销毁该连接。

至于本端为何迟迟收不到对端的 `FIN` 包，通常情况下都是因为对端机器出了问题，或者是因为太繁忙而不能及时 `close()`。所以，通常我们都建议将 `tcp_fin_timeout` 调小一些，以尽量避免这种状态下的资源开销。对于数据中心内部的机器而言，将它调整为 2s 足以：

```
net.ipv4.tcp_fin_timeout = 2
```

我们再来看 TIME_WAIT 状态，TIME_WAIT 状态存在的意义是：最后发送的这个 ACK 包可能会被丢弃掉或者有延迟，这样对端就会再次发送 FIN 包。如果不维持 TIME_WAIT 这个状态，那么再次收到对端的 FIN 包后，本端就会回一个 Reset 包，这可能会产生一些异常。

所以维持 TIME_WAIT 状态一段时间，可以保障 TCP 连接正常断开。TIME_WAIT 的默认存活时间在 Linux 上是 60s (TCP_TIMEWAIT_LEN)，这个时间对于数据中心而言可能还是有些长了，所以有的时候也会修改内核做些优化来减小该值，或者将该值设置为可通过 sysctl 来调节。

TIME_WAIT 状态存在这么长时间，也是对系统资源的一个浪费，所以系统也有配置项来限制该状态的最大个数，该配置选项就是 tcp_max_tw_buckets。对于数据中心而言，网络是相对很稳定的，基本不会存在 FIN 包的异常，所以建议将该值调小一些：

```
net.ipv4.tcp_max_tw_buckets = 10000
```

Client 关闭跟 Server 的连接后，也有可能很快再次跟 Server 之间建立一个新的连接，而由于 TCP 端口最多只有 65536 个，如果不去复用处于 TIME_WAIT 状态的连接，就可能在快速重启应用程序时，出现端口被占用而无法创建新连接的情况。所以建议你打开复用 TIME_WAIT 的选项：

```
net.ipv4.tcp_tw_reuse = 1
```

还有另外一个选项 tcp_tw_recycle 来控制 TIME_WAIT 状态，但是该选项是很危险的，因为它可能会引起意料不到的问题，比如可能会引起 NAT 环境下的丢包问题。所以建议将该选项关闭：

```
net.ipv4.tcp_tw_recycle = 0
```

因为打开该选项后引起了太多的问题，所以新版本的内核就索性删掉了这个配置选项：

🔗 [tcp: remove tcp_tw_recycle.](#)

对于 CLOSE_WAIT 状态而言，系统中没有对应的配置项。但是该状态也是一个危险信号，如果这个状态的 TCP 连接较多，那往往意味着应用程序有 Bug，在某些条件下没有调用 close() 来关闭连接。我们在生产环境上就遇到过很多这类问题。所以，如果你的系统中存在很多 CLOSE_WAIT 状态的连接，那你最好去排查一下你的应用程序，看看哪里漏掉了 close()。

至此，TCP 四次挥手过程中需要注意的事项也讲完了。

好了，我们这节课就到此为止。

课堂总结

这节课我们讲了很多的配置项，我把这些配置项汇总到了下面这个表格里，方便你记忆：

配置项	说明
net.ipv4.tcp_syn_retries	数据中心的网络质量都很好，如果Client得不到Server的响应，很可能是Server本身出了问题。在这种情况下，Client及早地去尝试连接其他的Server会是一个比较好的选择，所以重传次数可以少一些。
net.ipv4.tcp_syncookies	开启SYN Cookies可以防止部分SYN Flood攻击。
net.ipv4.tcp_synack_retries	与tcp_syn_retries是同样的思路，对于数据中心的服务器而言，不需要进行那么多的重传。
net.ipv4.tcp_max_syn_backlog	这一项是最多可以积压多少半连接。对于服务器而言，可能瞬间会有非常多的新建连接，所以我们可以适当地调大该值，以免SYN包被丢弃而导致Client收不到SYNACK。
net.core.somaxconn	这一项是最多可以积压多少全连接。与前一个配置项类似，在数据中心内部的服务器上，我们一般都会适当调大该值。
net.ipv4.tcp_abort_on_overflow	这样可以避免发送reset包给client，进而导致client出现异常。
net.ipv4.tcp_fin_timeout	迟迟收不到对端的FIN包，通常情况下都是因为对端机器出了问题，或者是太繁忙而不能及时close()，所以通常我们都建议将该值调小一些，以尽量避免这种状态下的资源开销。
net.ipv4.tcp_max_tw_buckets	减少该值来避免资源的浪费，对于数据中心而言，网络是相对很稳定的，所以我们一般都会调小该值。
net.ipv4.tcp_tw_reuse	主要用来解决，在快速重启应用程序时，出现端口被占用而无法创建新连接的情况。
net.ipv4.tcp_tw_recycle	这一项如果不配置为0，会容易引起一些异常。

当然了，有些配置项也是可以根据你的服务器负载以及 CPU 和内存大小来做灵活配置的，比如 tcp_max_syn_backlog、somaxconn、tcp_max_tw_buckets 这三项，如果你的物理内存足够大、CPU 核数足够多，你可以适当地增大这些值，这些往往都是一些经验值。

另外，我们这堂课的目的不仅仅是为了让你去了解这些配置项，最主要的是想让你了解其背后的机制，这样你在遇到一些问题时，就可以有一个大致的分析方向。

课后作业

课后请你使用 tcpdump 这个工具来观察下 TCP 的三次握手和四次挥手过程，巩固今天的学习内容。欢迎在留言区分享你的看法。

感谢你的阅读，如果你认为这节课的内容有收获，也欢迎把它分享给你的朋友，我们下一讲见。

提建议

更多课程推荐

程序员的数学基础课

在实战中重新理解数学

黄申

LinkedIn 资深数据科学家



涨价倒计时 🕒

今日秒杀 **¥79**，9月11日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 分析篇 | 内存泄漏时，我们该如何一步步找到根因？

精选留言 (4)

写留言



欧阳

2020-09-12

老师好：
FIN_WAIT_2 超时时间是 tcp_fin_timeout 控制，
TIME_WAIT 默认也是 60s，但是 /proc/sys/net/ipv4/ 下没有wait相关文件名，
TIME_WAIT 是与 FIN_WAIT_2 共用了同一个选项吗？

展开

3



许童童

2020-09-12

不错，老师总结得很好

展开



Su

2020-09-12

高产啊.

展开



青生先森

2020-09-12

又涨知识了

展开

